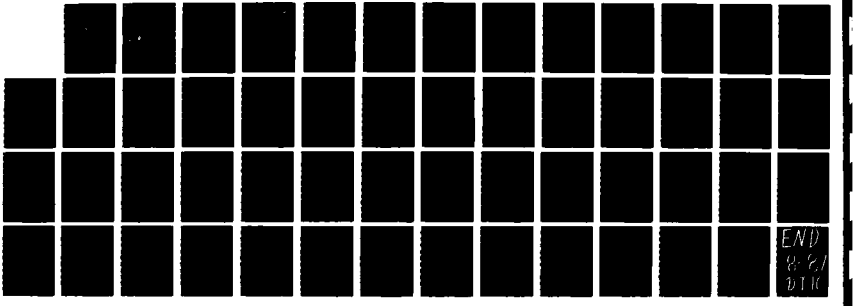
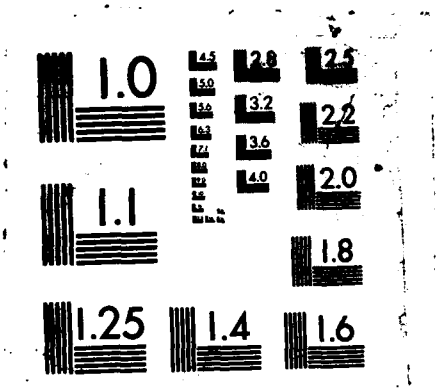
AD-A182 445 THE DEFINITION OF PRODUCTION QUALITY ADA* COMPILER(U) 1/1
AEROSPACE CORP EL SEGUNDO CA M O HOGAN ET AL
20 MAR 87 TR-0086A(2902-03)-1 SD-TR-87-29
UNCLASSIFIED F04701-85-C-0086 F/G 12/5 NL





(13)

REPORT SD-TR-87-29

The Definition of a Production Quality Ada* Compiler

Prepared by

M. O. HOGAN
Systems Software Engineering Department

E. P. HAUSER
Systems Software Engineering Department

S. M. MENICHELLO
Mission Software Department

20 March 1987

Prepared for

SPACE DIVISION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Station
P.O. Box 92960, Worldway Postal Center
Los Angeles, CA 90009-2960

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

This report was submitted by The Aerospace Corporation, El Segundo, CA 90245, under Contract No. F04701-85-C-0086-P00016 with the Space Division, P. O. Box 92960, Worldway Postal Center, Los Angeles, CA 90009-2960. It was reviewed and approved for The Aerospace Corporation by E. R. Frazier, Director, Systems Software Engineering Department, Engineering Group.

Mr. Giovanni Bargero, SD/ALR, approved the report for the Air Force.

This report has been reviewed by the Public Affairs Office (PAS) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

This technical report has been reviewed and is approved for publication. Publication of this report does not constitute Air Force approval of the report's findings or conclusions. It is published only for the exchange and stimulation of ideas.

Giovanni Bargero
SD/ALR

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS AD-A182 445
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-0086A(2902-03)-1	5. MONITORING ORGANIZATION REPORT NUMBER(S) SD-TR-87-29
6a. NAME OF PERFORMING ORGANIZATION The Aerospace Corporation Engineering Group 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Space Division
6c. ADDRESS (City, State, and ZIP Code) El Segundo, CA 90245-4691	7b. ADDRESS (City, State, and ZIP Code) Los Angeles Air Force Station Los Angeles, CA 90009-2960
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F04701-85-C-0086-P00016
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
The Definition of a Production Quality Ada Compiler

12. PERSONAL AUTHOR(S)
Hogan, Michael O.; Hauser, Elaine P.; Menichiello, Suzanne M.

13a. TYPE OF REPORT	13b. TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
		20 March 1987	47

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Ada compiler, selection; Ada compiler, procuring; Ada compiler, specifications; Ada compiler, evaluating; (cont.)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report specifies a set of minimal requirements that an Ada compiler must have to be considered production quality. Additional criteria are given for features that increase a compiler's usefulness for production use. The report can serve as a guideline for evaluating an existing Ada compiler for potential project use, or it can be used in preparation of specifications for procuring an Ada compiler.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

19 KEY WORDS (Continued)

20. ABSTRACT (Continued)

18. SUBJECT TERMS (continued)

Ada compiler, requirements; production quality Ada compiler; project Ada compiler

EXECUTIVE SUMMARY

This study, funded by Project Element 64740F, was conducted for Space Division's Directorate of Computer Resources, SD/ALR, to develop a quantifiable definition of the term "production quality" as applied to Ada compilers. This report specifies minimal and desirable requirements for the important characteristics of an Ada compiler that are beyond those required for validation, including performance, capacity, user-friendliness, reliability, documentation, and certain language features left to the discretion of the compiler builder.

The requirements given in this report are applied to the complete compiler system, which includes the compiler, the Ada library manager, linker/loader, and Ada run-time system. Certain requirements given are minimal criteria that all compilers must meet to be considered production quality, while other requirements give criteria for highly desirable features that the ideal production quality Ada compiler should have but which fall into a gray area when quantifying them. No method is given for how these additional criteria should be ranked in importance to rate a compiler, but by implication the closer a compiler meets each requirement, the better it is.

The following summarizes the minimal capabilities that an Ada compiler must have to be considered production quality in the areas of performance, quality assurance, and documentation.

The compiler must have a minimum compilation speed of 500 Ada source statements (essentially the number of executable statements plus the number of declarations) per minute of elapsed wall-clock time for a dedicated host computer that executes at a rate of 1 million instructions per second. This compilation speed is considerably less than would be expected from a compiler of an older programming language and should increase as Ada compilers mature. The compiled code produced by the compiler must occupy no more than 30%

additional space and take no more than 15% longer to execute than an equivalent hand-coded assembly language program.

Before being released for production use, the compiler must pass the required validation process and undergo a field evaluation period of 20 site-months (number of testing sites times months of testing). After release, the compiler must exhibit no more than 1 new error for each 250,000 new Ada source statements compiled.

Certain documents should be supplied by the compiler vendor: a User's Manual, Run-Time System Manual, Version Description Document, and Installation Manual should be delivered with the compiler. The Validation Summary Report from the Ada Validation Facility must be provided. Documents required by DOD-STD-2167 are required for compilers that are developed under contract to a DOD military program.

The following summarizes requirements in the gray area of highly desirable but less quantifiable features that a production quality compiler should have. These criteria deal with issues of user interface, implementation of language features, and compiler capacity limits.

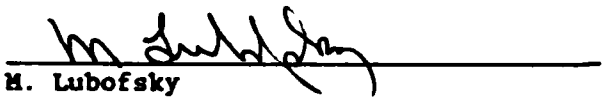
The compiler should provide the user with the option of performing syntax checking only, without producing object code. It should also be able to produce various Ada source code, cross reference, and assembly code listings at the option of the user. Diagnostic messages should explain errors clearly. The ability to interface with external tools is also required, e.g. compatibility with an existing symbolic debugger. While not part of the compiler system proper, these tools are necessary in a complete programming environment. Other characteristics are specified in the body of the report to ensure that the compiler is "user-friendly."

Certain desirable features of the Ada language are specified that are either not required by the Ada Language Reference Manual (ARM) or are left to be defined by implementors. Such features include the pragma INLINE, which causes the object code for a subprogram to be expanded inline at the point of call; representation specifications, which specify how the compiler is to represent certain program elements in the target computer hardware; and interfacing with other languages. The predefined data types provided by the compiler should use the full capabilities of the target computer. Additional implementation issues, such as how task scheduling and exception handling are done, are also specified.

The ARM does not specify capacity limits for the compiler, so minimal capacities that a production quality compiler should have are given in this report. A few of these are as follows: at least 1024 compilation units in a program, 4096 Ada source statements in a compilation unit, 120 characters in a source line, 4096 declarations in a compilation unit, 64 parameters in a subprogram, 32 dimensions in an array, 65,535 characters in a string, 65,535 elements in an array, 16 tasking priority levels, and 64 levels of block, loop, or subprogram nesting. Capacity requirements for other language elements are given in the report.

PREFACE

This study to define the requirements for a Production Quality Ada Compiler was conducted for Space Division's Directorate of Computer Resources, SD/ALR, as part of its participation in the Program Element 64740F technology transition effort. The results of this study, as contained herein, are intended to be used as guidelines for evaluating an existing Ada compiler or for selecting one of several potential compilers for project use. It will also be useful in preparing a specification for procuring a production quality Ada compiler. The conclusions drawn are not absolutes but rather are based in part on several reviews by a large number of compiler users and vendors, and on compiler statistics from validation summary reports.



M. Lubofsky
Systems Support Office
Computer Resources Management
and Standards Office

CONTENTS

CONTENTS (Continued)

SECTION 1
INTRODUCTION

Many times a compiler vendor or project manager will indiscriminately refer to a compiler as being "production quality," implying that this compiler is somehow of a superior quality to others. Yet little objective evidence is offered to justify the use of this term. When pressed for a definition of "production quality," little more is said than that the compiler is ready for use in a production environment. At one meeting, Lt. Col. Ed Koss, then Director of Computer Resources at Space Division, asked this very question about the Air Force Intermetrics Ada compiler. Not content with the usual vague reply, he proposed a project to develop a definition of a production quality Ada compiler; hence this report.

The purpose of this report is to provide a measurable definition of a production quality Ada compiler. It is intended that the criteria set forth here can be used to measure the validity of a vendor's claim of production quality for his Ada compiler.

One might question the need for such a definition when all Ada compilers must be validated for conformance to the Ada Language Reference Manual [ARM 83] (ANSI/MIL-STD-1815A) by an extensive suite of tests, the Ada Compiler Validation Capability (ACVC). However, by intent the ACVC does not test for many traditionally important characteristics of a production quality compiler, such as performance, capacity, user-friendliness, language tools interface, reliability, and user documentation. The quality of Ada compilers can vary further in areas of the language that are left as implementation defined by the ARM, or that are not required in order to pass validation. Thus, the task of defining a production quality Ada compiler presents the challenge of specifying language requirements beyond those required for validation as well as specifying the traditional areas common to all compilers.

Before defining the characteristics of a production quality Ada compiler, we must first define a compiler. Strictly speaking, an Ada compiler

is just one component of a larger system that includes the host operating system, the linker/loader, the Ada run-time system, and the target computer operating system. In addition, Ada's separate compilation feature requires a separate library manager to keep track of compilation units. Separate compilation causes some checking traditionally done by the compiler to be deferred to the linker/loader. Since some of the functionality usually associated with the compiler proper is now being distributed to these other components, the term compiler shall be taken to mean the compiler proper along with the Ada library manager, the linker/loader, the Ada run-time system, and any other component required to convert an Ada program from source code into an executable load module.

We specifically excluded from this definition any specification of host or target computer hardware and operating system, although the complete hardware and operating environment for both the host and target computers must be specified for each validated compiler.

The original goal of this report was to provide a minimal set of criteria that must be met by all production quality Ada compilers. In the process of researching such criteria, it became clear that the evaluation of a compiler can never be reduced to a simple formula. The evaluator must always make qualitative judgments about the relative importance of various aspects of the compiler. To assist in evaluating the importance of the requirements given in this report, we have divided them into two groups. The requirements of one group are considered to be fundamentally essential for a production quality compiler and must be met exactly. These requirements are designated by (M), for "minimal." The requirements of the other group, which are not marked by any symbol, are important compiler features for production use and should be met as closely as possible. The degree to which each of these requirements contributes to production quality is not clear. Eventually, these additional criteria might be weighted and used to give a compiler a numerical rating, although we have not done this due to the difficulty of just identifying the necessary requirements.

The Ada Programming Support Environment (APSE) Evaluation and Validation (E&V) Team is developing a quantitative and qualitative evaluation capability for Ada programming support environments, which include Ada compilers. Until such time as the E&V technology becomes available, this report should provide criteria to assess the quality of validated Ada compilers.

In specifying these criteria we have not sought fairness. If some compiler cannot meet a requirement, the overall impact on the user is that this compiler is less production quality than one that does, even if the reason has to do with hardware or operating system limitations. We are on the side of the compiler user and have tried to specify features that users want and need without dictating compiler design.

This report is divided into eight sections including this introduction. Sections 2 and 3 define the primary performance and capacity requirements for the compiler. Section 4 deals with the interaction of the compiler with the user. Section 5 concerns the interface of the compiler with external tools such as a symbolic debugger. Section 6 gives Ada language requirements. Section 7 states compiler quality assurance and reliability requirements, and Section 8 lists the required documentation. Most of the requirements are followed by a rationale that explains the requirement or gives some justification for making it. The following definition of an Ada source statement applies to the rest of this report.

1.1 Definition of an Ada Source Statement. An Ada source statement shall be defined to mean: a basic declaration, a record component declaration, a simple statement, a compound statement, an entry declaration, terminate alternative, WITH clause, USE clause, generic parameter declaration, proper body or body stub, representation clause, alignment clause, or component clause.

Rationale

The number of Ada source statements should be equivalent to the number of semicolons in a specified amount of code, provided that the semicolons in comments, and in the formal parts of subprogram declarations, entry

declarations, and accept statements are not counted. The number of Ada source statements as defined here eliminates confusion over multiple statements on a source line or a single statement occupying multiple lines. The presence or absence of comments has no effect on the number of source statements.

SECTION 2

PERFORMANCE REQUIREMENTS

After correctness, performance is the most important compiler selection criterion. Compiler performance refers to the execution speed of the compiler on the host computer and the memory size and execution speed on the target computer of the object programs generated by the compiler.

2.1 Benchmarks Used. All performance requirements of this section shall be met using the programs of the test suite formulated by the Performance Issues Working Group (PIWG) of the SIGAda Users' Committee.

Rationale

Compiler performance is measured by benchmark programs. In the absence of widely accepted and comprehensive benchmark programs, the requirements of this section shall measure the compiler's performance in compiling and executing the performance benchmarks distributed by the Performance Issues Working Group of the Users' Committee of the ACM Special Interest Group on Ada. Use of the ACVC tests was rejected as a basis for performance criteria because they are very short and compiler overhead time becomes disproportionate, all parts of the language are tested evenly without weighting according to actual usage, and very little computation is done.

2.2 Definition of Benchmark Test Units. The requirements in this section assume a single compilation unit without any context clauses (WITH clauses) or generic instantiations.

Rationale

Some users have reported that the compilation speed of some Ada compilers is much slower when generics or WITH clauses are used. Although this is an important area for future research, present experience is not sufficient to specify minimal rates for compiler speed in processing library units named in context changes and generic instantiations. For example, compilation units having many WITH statements may be testing the efficiency of the Ada library manager instead of the compiler itself.

2.3 Definition of Time Used. All speed requirements of this section shall be measured in terms of elapsed (wall-clock) time.

Rationale

The timing interval must be precisely defined. Two commonly used times are CPU execution time and actual elapsed (wall-clock) time. Elapsed time more accurately reflects the user's perception of compilation speed since it accounts for system overhead such as paging in a virtual environment. For multi-user host operating systems, it is necessary to perform the compilation speed testing in stand-alone mode with no other users on the system.

2.4 Host System Performance Requirements.

Compiler host system performance criteria are concerned with the requirements for host computer memory in executing the compiler and the compiler execution speed. No requirements on the amount of memory occupied by the compiler are given, since fast compilation speed and the production of efficient object code are of higher priority than conserving host memory, a decreasingly precious resource due to the use of virtual memory techniques and the decreasing cost per bit for memory chips.

2.4.1 (M) The compiler shall compile a syntactically and semantically correct Ada program of at least 200 Ada source statements at a rate of at least 200 statements per minute (elapsed time), for each 1 MIPS of rated processing speed of the specified host computer, while meeting the object code requirements in 2.5.1 and 2.5.2.

Rationale

This requirement essentially says that if the optimization necessary to meet the object code performance requirements in 2.5.1 and 2.5.2 is user selectable, the compilation speed requirement is relaxed when optimization is engaged (see the next paragraph).

2.4.2 (M) The compiler shall compile a syntactically and semantically correct Ada program of at least 200 Ada source statements at a rate of at least 500 statements per minute (elapsed time), for each 1 MIPS of rated processing speed of the specified host computer, in the absence of requirements on object code efficiency.

Rationale

It is often desirable to have fast compilation times during development, which speeds initial coding and debugging, while slower compilation times are acceptable for final performance tuning and testing. This requirement specifies the basic compilation rate that the compiler must meet. If optimization is user selectable, this requirement can be met with optimization turned off since it does not specify the efficiency of the object code produced. However, as with the previous requirement, it does require the object code to be generated correctly.

2.4.3 The compiler shall compile a syntactically and semantically correct Ada program of at least 200 Ada source statements at a rate of at least 1000 statements per minute (elapsed time), for each 1 MIPS of rated processing speed of the specified host computer, with no requirement to generate object code.

Rationale

Many times during early program development it is necessary to do many compilations to find all the language errors before the object code is generated. This requirement says the compiler should perform syntax (and possibly semantic) checking at least twice as fast as when also generating object code.

2.5 Target System Performance Requirements.

The requirements of this section are intended to ensure that Ada compilers produce code that is as memory efficient and as fast as code

produced by compilers for older high order languages. If Ada is to be widely used in embedded military systems, Ada compilers must be able to produce object code comparable in quality to compilers of existing high order languages, such as FORTRAN and JOVIAL J73. These requirements may not be directly applicable to compilers for some target computers that have a high-level "Ada code" type of instruction set architecture (e.g., the Rational R1000). But since such computers are not in widespread use, and since most computers used for military applications have conventional instruction sets, these requirements will apply to most compilers.

2.5.1 (M) The compiler shall produce an object code program that requires no more than 30% additional target computer memory space over an equivalent program written in assembly language.

2.5.2 (M) The compiler shall produce an object code program that requires no more than 15% additional execution time over an equivalent program written in assembly language.

Rationale

The efficiency of the code produced by a compiler is determined by the amount of target computer memory usage (object code size) and its execution speed. The standard for efficiency to which most compilers have been compared is hand-coded assembly language routines that implement the same algorithm as the higher-level language construct translated by the compiler. An equivalent assembly program is one that performs the same run-time checks (range checks, dereference checks, etc.) and uses the same run-time conventions as the Ada program. The equivalent assembly program should be straightforward and not resort to special tricks or tuning techniques such as changing an $O(N^2)$ algorithm to O(NlnN). Using an unvalidated Ada subset compiler for the Zilog 8000, McDonnell Aircraft Co. demonstrated a 1.6 memory expansion and 1.1 time expansion of Ada programs over equivalent hand-coded assembly programs used for the digital flight control system for the F-15 fighter aircraft.

SECTION 3
COMPILER CAPACITY REQUIREMENTS

Another important aspect of compiler performance is the capacity of the compiler to accept various sizes and ranges of Ada constructs in the Ada source program being compiled. The greater the capacity of the compiler, the less the programmer will be restricted in his use of the language. Although compilers hosted on or targeted to smaller computers (i.e., 16K-64K RAM) may not be able to achieve many of these requirements, the intent is that if a computer provides enough memory, the compiler should place a minimum of restrictions on the programmer.

Although a compiler's actual capacity limits may be larger than those specified here, the minimum values given represent our best judgment based upon experience with military embedded systems and upon three other sources: statistical data on existing Ada compilers, the consensus from a survey of Ada implementors and users, and comments from reviewers in the Ada community of an initial draft of this report.

3.1 Program Limitations. The compiler shall provide the following minimum capacities for each of the Ada program elements listed when provided with sufficient virtual storage:

library units in a program library	2048
compilation units in a program	1024
Ada source statements in a program	2,500,000
maximum size (in words) of a program	2,500,000
ELABORATE pragmas	512
width of source line (& length of identifier)	120

Rationale

The definition of Ada source statements given in Section 1.1 is used here. The width of an Ada source line is a compromise, since many terminals and punched cards limit source lines to 80 characters but most printers will display 133 characters. It is difficult to properly indent a program with fewer than 120 characters in a source line. The maximum program size is given in terms of the number of words of memory as defined for the target computer.

3.2 Compilation Unit Limitations. The compiler shall provide the following minimum capacities for each of the compilation unit elements listed when provided with sufficient virtual storage:

library units in a single context clause	16
library units WITHed by a compilation unit	256
external names	4096
Ada source statements in a compilation unit	4096
identifiers (including those in WITHed units)	4096
declarations (total) in a compilation unit	4096
type declarations	1024
subtype declarations of a single type	1024
literals in a compilation unit	1024

Rationale

With Ada's separate compilation feature, the need for very large compilation units is gone, and the need to support compilation units in excess of 4096 Ada source statements is questionable. The number of declarations allowed in a compilation should equal the number of identifiers allowed. A useful limit on these is the number of Ada source statements, allowing every source line to contain one declaration.

3.3 Program Unit Limitations. The compiler shall provide the following minimum capacities for each of the program unit (subprogram, package, task, or generic unit body) elements listed when provided with sufficient virtual storage:

depth of nesting of program units	64
depth of nesting of blocks	64
depth of nesting of case statements	64
depth of nesting of loop statements	64
depth of nesting of if statements	256
elseif alternatives	256
exception declarations in a frame	256
exception handlers in a frame	256
declarations in a declarative part	1024
identifiers in a declarative part	1024
frames an exception can propagate through	unlimited

Rationale

The depth of nesting of program units refers to the number of program units that may be declared within each other, not the total number of units that may be declared within a unit at the same level. An exception may be propagated through as many levels as necessary to reach the top of the calling chain. Any smaller number effectively limits the number of subprogram calls in a chain (including recursive calls). For example, if an infinite recursion causes a stack overflow, the resultant STORAGE_ERROR exception should be propagated to the outermost calling unit to allow recovery.

3.4 Task Limitations. The compiler shall provide the following minimum capacities for each of the task elements listed when provided with sufficient virtual storage:

values in subtype SYSTEM.PRIORITY	16
simultaneously active tasks in a program	512
accept statements in a task	64
entry declarations in a task	64
formal parameters in an entry declaration	64
formal parameters in an accept statement	64
delay statements in a task	64
alternatives in a select statement	64

Rationale

Although many tasks running at the same time might seriously impact performance, the number of active tasks allowed does not determine how many tasks can actually be running simultaneously, since tasks could be blocked awaiting rendezvous.

3.5 Subprogram Limitations. The compiler shall provide the following minimum capacities for each of the subprogram elements listed when provided with sufficient virtual storage:

formal parameters	64
levels in a call chain	unlimited

3.6 Package Limitations. The compiler shall provide the following minimum capacities for each of the package elements listed when provided with sufficient virtual storage:

visible declarations	1024
private declarations	1024

3.7 Statement Limitations. The compiler shall provide the following minimum capacities for each of the statement elements listed when provided with sufficient virtual storage:

declarations in a block	1024
enumeration literals in a single type	512
dimensions in an array	32
total elements in an array	65535
components in a record type	256
discriminants in a record type	64
variant parts in a record type	64
size of any object in bits	65535
characters in a value of type STRING	65535

Rationale

Although the predefined type STRING is declared in package STANDARD to be an array of characters with an index of type POSITIVE, which is a subtype of INTEGER from 1 to INTEGER'LAST, the ARM does not specify how the predefined type INTEGER is implemented (thus determining the value of INTEGER'LAST) or how many elements an array can have (thus, limiting the maximum number of characters in a value of type STRING).

3.8 Expression Limitations. The compiler shall provide the following minimum capacities for each of the expression elements listed when provided with sufficient virtual storage:

operators in an expression	128
function calls in an expression	128
primaries in an expression	128
depth of parentheses nesting	64

Rationale

Primaries are defined in ARM 4.4 and are essentially the elements, such as literals and variables, that are combined with operators to make up expressions.

SECTION 4

USER INTERFACE REQUIREMENTS

This section addresses the means by which the compiler elicits information from and returns information to the user. The requirements here attempt to define what is meant by a compiler that is "user-friendly."

4.1 User Inputs.

4.1.1 The compiler shall be invokable from either a batch file command or an interactive command.

Rationale

For compilers that run on interactive host computers, this facility saves setup time and simplifies maintaining special command files for each mode of invoking the compiler.

4.1.2 The compiler shall be sharable (re-entrant) by multiple users, if the host operating system supports multiple users.

Rationale

It is possible, but unlikely, that a vendor would build a compiler which runs under a multi-process, multi-user host operating system but which can only be accessed by one user at a time.

4.1.3 The compiler shall implement options to perform the same function as pragmas SUPPRESS and OPTIMIZE.

Rationale

Command level options that perform the same function as the SUPPRESS and OPTIMIZE pragmas ease program development and testing since the source code does not have to be altered to insert and later remove these pragmas.

4.1.4 The compiler shall implement an option to recover from non-fatal errors as defined in 4.3.3. The recovery action taken shall be identified.

Rationale

The user should have the option to allow compilation to continue in the presence of non-fatal errors. For example, a compiler can usually tell if a semicolon is missing, which is not a fatal error. In this case, recovery should be attempted and compilation should continue. On the other hand, if a package specification named in a WITH clause cannot be found, this is a fatal error and compilation should be terminated. Meaningful messages identifying the action taken are necessary.

4.1.5 The compiler shall implement an option to disable the generation of diagnostic messages of a specified severity level.

Rationale

The user should have the flexibility to turn off output of certain levels of diagnostic messages when there is an awareness of the condition or a desire to focus on other compilation outputs. For some implementations, this option can provide an increase in compilation speed when enabled.

4.1.6 The compiler shall implement an option to select or suspend the generation of object code and/or assembly code.

Rationale

This requirement provides the user with the capability to perform syntax (and possibly semantic) checking, which most compilers can do quite rapidly, during the early stages of development without the time-consuming task of object code production.

4.2 Compiler Listings.

4.2.1 The compiler shall be able to produce at the option of the user a compilation listing showing the source code with line numbers.

4.2.2 The compiler shall be able to produce at the option of the user a list of diagnostic messages either at the position in the source code where the condition occurred, and/or at the end of the compilation listing, even if the compilation terminates abnormally.

Rationale

Listings that have embedded diagnostic messages, rather than an error message summary at the end of the listing, ease the task of debugging. As this is sometimes a matter of programmer preference, a good compiler should give the user the option of embedded or last page diagnostic messages, or both.

4.2.3 The compiler shall be able to produce at the option of the user an assembly or pseudo-assembly output listing.

4.2.4 The compiler shall be able to produce at the option of the user an assembly or pseudo-assembly output listing with embedded Ada source statements adjacent to the assembly code they generated.

Rationale

Assembly or pseudo-assembly code listings are invaluable in evaluating compiler errors and areas of inefficient code production. An assembly listing with embedded Ada code is essential in developing applications for embedded target computers, where testing, debugging, and maintaining code must be done at the assembly language or machine language level. For highly optimizing or interpretive compilers this may not always be possible.

4.2.5 The compiler shall be able to produce at the option of the user a cross reference (set/use) listing.

4.2.6 The compiler shall be able to produce at the option of the user a map of relative addresses of variables and constants.

Rationale

These listings are used for debugging and testing and have traditionally been included in compilation listings. They are invaluable for developing and testing software for embedded systems.

4.2.7 For each compilation, the compiler shall be able to produce at the option of the user a statistics summary listing with the following information:

a. Number of statements
b. Number of source lines
c. Compile time per program module (CPU time)
d. Total compile time (CPU and elapsed time)
e. Total number of instructions generated
f. Total number of data words generated
g. Total size of object module generated

Rationale

This information is useful for determining compiler performance and for project management information. Some of this information could be supplied by external tools, but all of it is available at compile time.

4.2.8 All listings shall include the following header information on every page:

a. Date and time of compilation
b. Compilation unit name
c. Type of listing
d. Page number within total listing
e. User identification

Rationale

Header summary information is necessary to easily identify listings.

4.2.9 All listings shall have the following additional information within the listing:

a. Compiler name, version number, release date
b. Host and target computer configurations
c. Specified and default control options

d. Source file name

e. Object file name

Rationale

This information is necessary but need not be included on every page. It can be on the first page or included with the compilation summary at the end of the listing.

4.3 Diagnostic Messages.

4.3.1 Each diagnostic message shall contain the message text, a reference number for additional information in the compiler documentation, and a severity level.

4.3.2 The diagnostic message text shall be sufficiently informative to enable the user to analyze the problem without consulting compiler documentation.

4.3.3 The severity levels of diagnostic messages shall include the following error classes:

a. Note: Information to the user; the compilation process continues and the object program is not affected.

b. Warning: Information about the validity of the program. The source program is well-defined and semantically correct; the object program may not behave as intended.

c. Error: An illegal syntactic or semantic construct with a well-defined recovery action. Compilation continues and the object program contains code for the illegal construct; the object program may behave meaninglessly at run-time.

d. Serious Error: Illegal construct with no well-defined recovery action. Syntax analysis continues but no object program is generated.

e. Fatal Error: Illegal construct with no reasonable syntactic recovery action. Compilation terminates and no outputs other than the source listing and diagnostic messages are produced.

Rationale

The method of presentation of diagnostic messages is an essential part of the user interface. Messages should be classed by their level of severity, e.g. fatal, serious, recoverable, warning, or note. The names of the classes used here are suggestive; it is not required that the same names or diagnostic codes be used. It is necessary that the compiler recognize and diagnose these classes of messages.

4.3.4 The compiler shall issue a diagnostic message to indicate any capacity requirements that have been exceeded.

Rationale

The compiler shall indicate whenever any of the capacity requirements of section 3 have been exceeded.

4.3.5 The compiler shall not abort regardless of the type or number of errors encountered.

Rationale

Even if the compiler encounters an overwhelming number of errors, it should quit gracefully and produce a listing of the errors encountered at the point where it could no longer continue. A compiler that simply aborts when it cannot compile a program with too many or certain types of errors is extremely frustrating to use, since no listings are produced.

SECTION 5

EXTERNAL TOOLS INTERFACE REQUIREMENTS

This section concerns the compiler's ability to interface with tools that may be external to the compiler but are generally considered necessary in a complete programming support environment. Although specifying the exact functionality of these tools is outside the scope of this document, it is necessary that a production quality compiler have the proper interfaces to work with the tools specified here. These interfaces can be either proprietary, working only with tools developed by the compiler vendor, or they can be open and work with a number of tools. The list here is minimal and expected to grow over time.

5.1 Listing Tools.

5.1.1 The compiler and/or external tool shall be able to produce a source listing with indentations to show control constructs.

Rationale

Indented source listings, sometimes called "pretty-printed" listings, are considered by many users to be useful in understanding the flow of control of a program. This capability permits any user to obtain an indented source listing without relying on the programmer to indent his own code as the program is developed.

5.1.2 The compiler, linker/loader, and/or external tool shall be able to produce an absolute assembly code listing.

Rationale

An absolute assembly listing shows the absolute memory location in the target computer of each machine/assembly instruction. Depending on the application (e.g., embedded avionics and spaceborne applications involving formal IV&V), an absolute assembly listing can be essential. This listing may be produced by the compiler or by special tools that interface with the compiler and linker/loader.

5.1.3 The compiler and/or library manager shall be able to produce at the option of the user a dependency listing showing which library units are WITHed by other units.

Rationale

This listing is useful when modifying or debugging a large program. It allows tracing units that reference a particular library unit so that modifications can be made in the user program units. This listing can be produced by the compiler, the library manager, or by a separate programming environment tool.

5.1.4 The compiler and/or library manager shall have the capability of listing all out-of-date (obsolete) library units with the option of selectively recompiling such units before linking.

Rationale

Ada's separate compilation mechanism requires a special Ada library manager function with its own particular production quality requirements. Compilation units which depend upon a particular program unit that has been changed and recompiled also need to be recompiled before linking and execution. With large program units, controlling these dependencies and recompilations by hand is unreasonably time-consuming.

5.2 Linker/Loader.

5.2.1 The compiler and/or linker/loader shall include in the load module only those subprograms that are actually referenced by the object program.

Rationale

A program unit that WITHs a package in order to access several subprograms should not have to pay the penalty of having every subprogram in the package included by the linker in the load module; only the code for subprograms actually referenced should be included. This feature is necessary in the case of a very large library package, such as a vendor supplied math package, in order to reduce the size of the load module.

5.2.2 The compiler and/or linker/loader shall include in the load module only those run-time system modules that are referenced by the object program.

Rationale

The run-time system is linked with the object code to produce the load module. Only the parts of the run-time system actually referenced should be included in the load module. Otherwise the load module size is increased unnecessarily, which may be unacceptable for some embedded systems. For example, an application that does not use tasks should not incur the space penalty of the added run-time code that handles tasking.

5.2.3 The compiler and/or linker/loader shall support the partial linking of object modules as specified by the user.

Rationale

The user should have the option of selecting which external names will remain external in the partially linked sets. A partially linked object module must then be acceptable as input for subsequent linking.

5.2.4 The compiler and/or linker/loader shall support the linking of designated object modules without including them in the load module.

Rationale

This requirement allows linkage to a "phantom" load module which can be separately down-loaded into known absolute locations. In this way linkage to shared resident code, which could be located in ROM, is allowed.

5.3 Symbolic Debugger. The compiler shall be able to produce object code files and other types of data necessary to debug those files with an available source-level (symbolic) debugger.

Rationale

It is desirable that a production quality compiler system provide a symbolic debugger capability. Therefore, the compiler should provide for the necessary

interfaces to work with a symbolic debugger available on the same host computer as the compiler. An example of this interface is the saving of the symbol table and intermediate language information.

SECTION 6

ADA LANGUAGE REQUIREMENTS

Although conformance to the Ada standard (ANSI/MIL-STD-1815A)[ARM83] is the foremost functional requirement of the compiler, the ARM intentionally leaves the interpretation of many language features to the compiler implementor. These areas of variability are generally limited to pragmas, attributes, certain machine-dependent conventions, and certain allowed restrictions on representation clauses. This section specifies the preferred implementation of these language features.

6.1 General. The compiler shall eliminate statements or subprograms that will never be executed (dead code) because their execution depends on a condition known to be false at compilation time.

Rationale

The ARM permits a compiler to eliminate code that it can determine will never be executed in order to provide a conditional compilation facility within the language. By use of this feature, sections of the code will be eliminated if some static expression whose value is known at compile time (ARM 4.9) is set to a certain value (e.g., "DEBUGGING : CONSTANT BOOLEAN:=FALSE"). Thus, the programmer can leave diagnostic code (e.g., text I/O statements used for monitoring errors during development) in the program without incurring a size penalty in the production version of the object code.

6.2 Character Sets.

6.2.1 The compiler shall allow the Ada program text to contain any of the 95 graphic characters and 5 form effectors of the ISO 7-bit character set (ISO Standard 646) to the extent supported by the host computer.

Rationale

The ARM (section 2.1) only requires a compiler to recognize a set of 56 basic graphic characters, since some older computers cannot support the full 95 character set. A production quality compiler should also allow the 26 lower case letters and 13 other special characters.

6.2.2 The predefined packages TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO shall support input and output of data containing any of the 128 ASCII character literals of the predefined type STANDARD.CHARACTER.

6.2.3 The compiler shall allow comments and values of the predefined type STRING to contain any of the 128 ASCII characters contained in the predefined type STANDARD.CHARACTER.

Rationale

These requirements allow writing programs that generate special escape codes and control characters used to drive external devices (e.g., a graphics terminal).

6.3 Data Representation.

6.3.1 The compiler shall provide predefined types in package *STANDARD* for all the integer and floating point types provided by the target computer.

Rationale

The compiler should provide access to the full capability of the data types implemented by the target computer instruction set architecture. For example, the predefined integer types provided by a computer might vary from 8 bits to 64 bits or longer, and predefined floating point types could vary from 32 bits to 128 bits. Accordingly, the compiler should contain corresponding predefined types in package STANDARD, e.g., SHORT_INTEGER or LONG_FLOAT.

6.3.2 The compiler shall support universal integer calculations requiring up to 64 bits of accuracy.

Rationale

The ACVC has four tests that check whether or not a compiler can perform calculations on universal integers requiring an accuracy of 32 and 64 bits, i.e., on values that exceed SYSTEM.MAX_INT. Many compilers have passed the test for accuracy of 64 bits.

6.3.3 The components of array types with BOOLEAN components named in a pragma PACK shall be stored in contiguous memory bits, i.e., each component shall occupy only one bit of storage.

Rationale

Using instructions for shifting data, it is possible to pack any array or record so its components are stored at the next available bit location (i.e, aligned on bit boundaries instead of on byte or word boundaries).

6.3.4 The compiler shall support address clauses.

Rationale

Address clauses are desirable for small embedded computer applications for handling memory-mapped I/O (i.e., using a memory address to specify I/O ports or registers of devices), connecting to hardware interrupts, communicating with non-Ada routines, storing objects into particular types of memory (e.g., EPROM), and accessing specific locations to perform hardware checks. Address clauses may be impossible to support under virtual memory operating systems.

6.3.5 The compiler shall support length clauses, enumeration representation clauses, and record representation clauses.

Rationale

These clauses (ARM, Chapter 13) are needed to control the physical implementation of data where memory conservation is at a premium or where machine interfaces require a specific layout, e.g., I/O devices and telemetry words.

6.3.6 The range of integer code values allowed in an enumeration representation clause shall be MIN_INT to MAX_INT.

Rationale

This requirement allows a programmer to take advantage of the full range of integer values provided by the target computer to specify the hardware mapping of enumeration types. This is often necessary for systems programs that must interface with device drivers and other machine-dependent values.

6.3.7 The compiler shall allow non-contiguous integer code values in an enumeration representation clause.

Rationale

In systems programming, it is convenient to use enumeration literals for unique codes that must be passed to existing operating system routines and external interfaces. This requirement is tested by the ACVC tests and has been met by many Ada compilers.

6.3.8 The compiler shall support the SIZE attribute designator for enumeration types named in a length clause.

Rationale

The size attribute in a length clause allows a programmer to specify an upper limit on the number of bits to be allocated to objects of a designated enumeration type. For example, a BOOLEAN object can be represented using 1 bit, but for accessing efficiency an implementation might use an 8-bit byte representation. If space is a high priority, a programmer may force the compiler to use just 1 bit by using a length clause. (Since hardware defined integer types are likely to have faster built-in accessing and arithmetic instructions, this would probably result in slower program execution.)

6.3.9 The compiler shall support the SMALL attribute designator for fixed point types.

Rationale

The attribute SMALL is necessary to control the amount of storage allocated to fixed point types in order to improve storage efficiency or interface correctly with an external hardware device.

6.3.10 Memory space for the creation of objects designated by an access type shall not be allocated until allocators (new statements) for that type are executed.

Rationale

Not allocating blocks of memory for creating pools of access objects until they are needed minimizes object code size.

6.4 Subprograms.

6.4.1 The compiler shall expand inline any subprogram or generic subprogram instantiation that is named in a pragma INLINE and that meets the criteria of 6.4.2.

6.4.2 A subprogram or generic subprogram instantiation is a candidate for inline expansion if it meets the following criteria:

a. Its body is declared in either the current unit or the compilation library.

b. Its parameters or result type (for functions) are not task types, composite types with task type components, unconstrained array types, or unconstrained types with discriminants.

c. It does not contain another subprogram body, package body, body stub, generic declaration, generic instantiation, exception declaration, or access type declaration.

d. It does not contain declarations that imply the creation of dependent tasks.

e. It does not contain any subprogram calls that result in direct or indirect recursion.

6.4.3 The compiler shall expand inline any subprogram that meets the requirements in 6.4.2 and that is called only once.

Rationale

Inline expansion is an important feature to improve execution speed at the expense of increased object code size. Inline expansion of certain small subprograms or those called only once can save considerable time during

run-time execution with little or no increase in memory over the usual prologue and epilogue generated for a normal call.

6.4.4 The compiler shall provide the capability for main subprograms to return a value to the target computer run-time system indicating the completion status of the program.

Rationale

This capability can be accomplished by allowing library units that are parameterless functions as well as procedure subprograms to be main programs. This capability simplifies the implementation of run-time executives for embedded systems.

6.5 Tasking.

6.5.1 The compiler shall provide a capability for handling target computer hardware or operating system interrupts as calls to Ada task entries.

Rationale

This capability might be provided by allowing the use of an address clause for associating a task entry with a hardware interrupt (ARM 13.5.1), or by providing pragmas to cause task entries to be associated with target operating system or computer hardware interrupts.

6.5.2 The execution-time overhead to perform a context switch or to terminate or abort a task shall be no more than that required to call or return from a subprogram.

Rationale

Once a task is established, the overhead to switch tasks should be equivalent to the time required to make a subprogram call.

6.5.3 The ordering of select alternatives in a selective wait statement shall not impact the execution speed of the program.

Rationale

Select alternatives include accept statements, delay statements, and terminate statements. The ARM does not define the mechanism for selecting one of several open alternatives when a rendezvous with any of them is possible. The intent of this requirement is that the mechanism chosen by the compiler's run-time system is fair (e.g., the first alternative is not always selected). This requirement can be tested simply by reordering the select alternatives of a typical program, recompiling it, and timing its execution.

6.5.4 The compiler shall dispatch the execution of ready tasks in a manner that will give each task an equal share of the processing resources consistent with any PRIORITY pragmas.

Rationale

The ARM does not prescribe a particular dispatching policy. The policy chosen by the implementor should be fair and not cause task starvation. For example, an easy but potentially unfair dispatching policy is to allow tasks to run until blocked. With this approach, a task containing a loop with a large number of iterations could run indefinitely while others would remain blocked.

6.5.5 Tasks that are blocked, completed, terminated, or not activated shall not impact the performance of the active tasks.

Rationale

A task is blocked if it is waiting for a rendezvous. Tasks are completed when they have reached the end of their executable statements. A task can become blocked while waiting for termination of dependent tasks. Idle tasks such as these should not consume processing resources until they become active.

6.5.6 The value of DURATION'DELTA shall not be greater than 1 millisecond.

Rationale

DURATION'DELTA is the smallest decimal increment of time (in seconds) that the compiler can support in a delay statement. The ARM requires DURATION'DELTA to be less than 20 milliseconds and recommends less than 50 microseconds.

Representation of 20 milliseconds (.020 decimal) requires 6 bits to the right of the binary point to represent, while 1 millisecond requires 10 bits. The ARM also requires that DURATION'LARGE be at least 86400 (the number of seconds in 1 day), which requires 17 bits to the left of the binary point. Together these requirements require an implementation to support fixed point types with a representation of at least 24 bits (6+17+sign bit) for DURATION'DELTA of 20 milliseconds and 28 bits for a DURATION'DELTA of 1 millisecond.

6.6 Exceptions.

6.6.1 An exception shall not impact execution speed until it is raised.

Rationale

Exceptions should be part of the termination actions of the frame they reside in rather than part of normal execution. It is important that exceptions which are not used do not slow execution of the program.

6.6.2 The compiler shall provide the pragma SUPPRESS or an equivalent capability to permit suppression of all predefined run-time checks in a designated compilation unit.

Rationale

To increase execution speed of the delivered program it may be necessary to turn off Ada's extensive run-time checking mechanism. The granularity at which execution of run-time checking can be disabled should be minimally at the compilation unit level (e.g., allow all checking to be disabled for all entities within a compilation unit). Pragma SUPPRESS provides a finer degree of granularity by allowing specific checks on specific entities to be disabled over any declarative region. A vendor may choose to implement pragma SUPPRESS or use another mechanism to achieve this capability.

6.6.3 The compiler shall issue a warning message to indicate static expressions that will always raise a constraint exception at run-time.

Rationale

The ARM does not require a compiler to detect that a particular construct will always raise a run-time exception. Static expressions (which are necessarily determinable at compile time) that always raise a constraint exception can be flagged at compile time. The programmer should not have to wait until run-time to learn about a constraint error.

6.7 Generics.

6.7.1 The compiler shall share code between multiple instantiations of generic units that do not differ in their underlying machine representation.

Rationale

When a generic subprogram is instantiated with different types which have identical machine representations, the compiler should recognize that the code produced by these various instantiations is identical and therefore make all references to a single copy of the generic unit, thereby minimizing the program memory usage.

6.7.2 The compiler shall allow generic specifications and bodies to be compiled in completely separate compilations.

6.7.3 The compiler shall allow subunits of a generic unit to be separately compiled.

Rationale

These requirements simply require that Ada's separate compilation semantics apply to generic units as well as subprograms and packages. The ACVC tests for these conditions.

6.8 Interface with Other Languages.

6.8.1 The compiler shall provide the pragma INTERFACE to allow importing of assembly language programs already assembled into the object code format of the target computer. The machine language interface for procedure and function parameters and function result types shall be documented.

Rationale

It is frequently necessary in embedded systems to implement many actions in machine language to meet timing and sizing requirements. The ARM suggests two mechanisms for doing this. One is to supply a predefined package called MACHINE_CODE for each target machine, which allows a programmer to make machine code insertions. The other is to provide the pragma INTERFACE for importing procedures and functions of the target machine assembly language. The latter is regarded by many as the safest way to interface assembly/machine language subprograms with Ada code. The exact nature of the Ada/machine language interface shall be documented in the User's Manual or Run-time System Manual (see 8.3 and 8.4)

6.8.2 The compiler shall provide the pragma INTERFACE, or an equivalent mechanism, to allow incorporation of subprogram bodies compiled from the standard system or application languages of the target computer.

Rationale

The programming languages that need to be interfaced with by use of pragma INTERFACE will vary with the target computer. For UNIX systems, the ability to interface with C programs is required; for many systems it is FORTRAN; for 1750A compilers it is JOVIAL. Reuse of the extensive body of well-tested code for mission critical applications written in FORTRAN and JOVIAL J73 (especially for Air Force projects) within newly developed Ada programs will be necessary to achieve a cost effective transition to Ada for existing DOD projects.

6.9 Unchecked Programming. The generic library subprograms UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION shall be implemented with no restrictions except that both objects in an unchecked conversion may be required to be of the same size.

Rationale

The ARM permits an implementation to place restrictions on unchecked conversions. Minimally such restrictions should be limited to requiring both to be of the same size (i.e., same number of bits). For types of different sizes, one approach is to truncate the high-order bits if the source type is larger than the target, and extend it with zero bits if it is smaller than the size of the target type.

6.10 Input/Output.

6.10.1 An implementation shall provide packages to allow input and output of FORTRAN-formatted text files for each target computer that supports text input/output.

Rationale

Many programmers regard format-directed input/output facilities (as provided by FORTRAN) as necessary for many data processing applications. This requirement is in addition to the TEXT_IO package required by the ARM. Compilers that only generate code for embedded computers with no text input/output capability need not provide this capability.

6.10.2 Package SEQUENTIAL_IO and package DIRECT_IO shall be able to be instantiated with unconstrained array types or with unconstrained record types which have discriminants without default values.

Rationale

In processing files with variable-sized records or arrays it is convenient to open them with a single unconstrained type, read in the value, and then test its size to determine how to process it. The ACVC tests this requirement.

6.10.3 The compiler shall allow more than one internal file to be associated with each external file for DIRECT_IO and SEQUENTIAL_IO for both reading and writing.

6.10.4 The compiler shall allow an external file associated with more than one internal file to be deleted.

Rationale

In writing generalized file processing programs it is convenient to be able to refer to the same file using different file names and to be able to delete such a file. The ACVC checks these requirements, which have been met by many compilers.

6.11 System Information.

6.11.1 The named numbers defined in package SYSTEM shall not limit or restrict the inherent capabilities of the target computer hardware or operating system.

Rationale

The value of STORAGE_UNIT should not be less than the number of bits in the smallest addressable storage unit, and the value of MEMORY_SIZE should not be less than the maximum number of addressable memory units. MIN_INT (MAX_INT) should be equal to the most negative (most positive) value of all the predefined integer types provided by the target computer hardware. MAX_DIGITS should not be less than the largest number of significant decimal digits in the mantissa of the largest floating point type provided by the target computer. MAX_MANTISSA should not be less than the largest number of binary digits of the mantissa of any fixed point hardware type, or equal to the maximum number of bits (not counting sign) of the largest integer type if fixed point is handled via integer representation. TICK should be equal to the smallest timing increment provided by the target computer hardware or executive services, if any.

6.11.2 The enumeration type NAME defined in Package SYSTEM shall have values for all target computers for which the compiler generates code.

Rationale

The constant SYSTEM_NAME in package SYSTEM takes on values of type NAME and can be used to write portable Ada code that can be tailored to a particular target computer depending on the value of SYSTEM_NAME, which can be set via the pragma SYSTEM_NAME.

6.12 Pragmas. An implementation shall provide the predefined pragmas CONTROLLED, ELABORATE, LIST, MEMORY_SIZE, OPTIMIZE, PAGE, STORAGE_UNIT, and SYSTEM_NAME.

Rationale

The listed pragmas can be implemented by most computer systems and should be implemented as defined in the ARM.

SECTION 7

QUALITY ASSURANCE AND RELIABILITY REQUIREMENTS

This section defines requirements for testing, configuration management, and maintenance of a compiler. These requirements are intended to assure that a compiler performs reliably.

7.1 (M) Validation. The compiler shall be validated by an Ada Validation Facility established and operated under the direction of the DOD Ada Joint Program Office in all configurations necessary to meet the requirements of this document.

Rationale

To be validated an Ada compiler must successfully pass the current version of the Ada Compiler Validation Capability test suite. Policies and procedures of the validation process are still undergoing change. The prospective compiler buyer should inquire as to the status of the policies and procedures for validation at the time of procuring an Ada compiler. The compiler should be validated with certain optional features, such as optimization, both on and off.

7.2 (M) Field Testing. The compiler shall be subjected to a minimum of 20 site-months of independent evaluation and usage in a realistic production work environment before release for production use.

Rationale

Field testing by production oriented users is necessary to find errors not discovered through validation and vendor testing. The measure selected for field testing is site-months (number of user organization sites times the number of months of testing). For example, 10 sites for 2 months, or 5 sites for 4 months is 20 site-months. To achieve the 20 site-month requirement, at least 3 sites should be used (with 7 months), or at least 2 months at 10 sites. A period of 20 site-months was felt to be an acceptable minimum based upon experience with existing Ada compilers.

7.3 Maintenance. Provisions for on-going problem correction of the compiler shall be provided.

Rationale

On-going problem correction is necessary for any complex software product; ideally, a production quality compiler should require little of this. The procuring users' agency should require some form of maintenance contract from the compiler vendor. If adequate maintenance capabilities cannot be provided, then the source code, development tools, maintenance documentation, data rights, and a set of regression test cases must be obtained by the procuring users' agency, e.g., the program office.

7.4 Configuration Management. The maintaining organization shall provide configuration management for the compiler, including maintenance of an up-to-date data base of compiler errors showing the nature and status of each error.

Rationale

Configuration management of changes to the compiler is important. The vendor or the users' agency responsible for compiler maintenance (e.g., the Government in the case of GFE compilers) must provide the necessary configuration management.

7.5 (M) Error Rate. The production quality compiler should exhibit an error rate of no more than 1 verified new error for each 250,000 new lines of Ada compiled. This rate shall decrease over time as the compiler matures.

Rationale

Industry experience with other mature HOL compilers indicates that an error rate exceeding this level is inadequate for production work. Although this requirement is difficult to verify and introduces other ambiguities (e.g., what constitutes an error), we felt that maturity and reliability are such important characteristics in a production quality compiler that to refrain from stating some minimal requirement would be unsatisfactory.

SECTION 8

DOCUMENTATION REQUIREMENTS

This section describes documents that should accompany a production quality compiler. Except for the documentation that must be in a specific format to meet DOD requirements, the documents in this section may be supplied in any format agreed upon by the vendor and the compiler buyer. Specifically, the documents may be combined in any appropriate manner, provided they contain the specified information.

8.1 (M) Validation Summary Report. The vendor shall provide a copy of the most recent version of the official validation summary report prepared by the Ada Validation Organization that validated the compiler. This report shall include both CPU and elapsed times required to run the ACVC tests.

Rationale

The Government (i.e., the Ada Joint Program Office and the Ada Validation Organization and its facilities) is not required to make this report publicly available. Therefore, the vendor must provide this report to the compiler purchaser.

8.2 (M) Ada Language Reference Manual (ARM). The compiler vendor shall supply a copy of the Ada Language Reference Manual (ARM) (ANSI/MIL-STD 1815A) that includes implementation-specific details of the compiler where applicable.

Rationale

Programmers should have readily available information about how the compiler implements parts of the Ada language which are implementation-defined in the ARM. In particular, the ARM requires that the reference manual of each Ada implementation include an appendix (Appendix F) describing all implementation-dependent characteristics.

8.3 (M) User's Manual. The vendor shall provide a User's Manual that describes how to use the compiler to develop Ada applications programs, including information on how to run the compiler. It shall include all system-dependent forms implemented in the compiler (i.e., machine-specific functions), methods of selecting debug aids, compiler options and parameters, and a complete list of error and warning messages provided by the compiler, with a description of each. Message descriptions shall reference the relevant section of the ARM. The manual shall include examples of the commands used to invoke the compiler and linker/loader system with various combinations of compiler and linker options, respectively.

Rationale

The purpose of the User's Manual is to provide information to programmers on how to use the compiler. It should contain all the information specific to the compiler being procured.

8.4 (M) Run-time System Manual. The vendor shall provide a Run-time System Manual for each target computer.

Rationale

The Run-time System Manual is necessary for the user who must understand how the run-time system is organized. This manual shall describe the details of the run-time system: each module, the function of each module, language written in, called by, modules called, design details, and run-time performance in terms of memory and execution speed.

8.5 (M) Version Description Document(VDD). The vendor shall provide a Version Description Document for each compiler configuration.

Rationale

The Version Description Document shall completely describe the host computer hardware and the host computer operating system (if any) including vendor, version, and configuration. The VDD shall also describe the format and type of code generated (i.e., whether assembly or object code), the exact target

computers for which code is generated, including instruction set architecture extensions/subsets, channels implemented, memory present, vendor of computer, model number, peripherals and options, and the exact target computer operating system (if any) including vendor, version, and configuration.

8.6 (M) Installation Manual. The vendor shall provide a detailed Installation Manual and all the necessary software materials for installing each host configuration of the Ada compiler, including several sample Ada programs with correct output.

Rationale

This manual shall provide complete information on the process of installing the compiler on the user's host computer system. This information should include the file structure of the compiler, how the compiler is supplied, what steps are necessary to install the compiler, and whether the user or the vendor shall perform the installation. Necessary software materials include prepared command files that instruct the host computer operating system to automatically install the compiler, and sample programs to allow the installer a means for verifying that the compiler has been properly installed. The Installation Manual may be combined with the User's Manual described above.

The following requirements are only necessary when procuring a production quality Ada Compiler developed under a DOD contract.

8.7 Maintenance Manual. The vendor shall provide a Maintenance Manual which presents the methods to be used in the general maintenance of all parts of the compiler. All major data structures, such as the symbol table and the intermediate language, shall be fully described. All debugging aids that have been inserted into the compiler shall be described and their use fully stated. If the compiler has a special "maintenance mode" of operation to assist in pinpointing errors, this shall be fully described.

Rationale

The Maintenance Manual is essential if the customer procuring the compiler intends to perform maintenance himself. The Maintenance Manual highlights the detailed design and coding information usually contained in the Software Product Specification. It should cover both code and data design.

8.8 (M) Software Product Specification. The vendor shall provide a Software Product Specification for the compiler in accordance with DOD-STD-2167 and Data Item Description DI-MCCR-80029.

Rationale

This document does not apply to off-the-shelf compilers, but it is required when the compiler is newly developed for a DOD contract. It contains design documentation, upgraded completion of the information in the Software Top Level Design Document and Software Detailed Design Document, and code listings for each unit in the compiler.

REFERENCES

[APSE84] Evaluation and Validation Team, Requirements for Evaluation and Validation of Ada Programming Support Environments, Version 1.0, (17 October 1984).

[D2167] Military Standard, Defense System Software Development, DOD-STD-2167 (4 June 1985).

[ARM83] Military Standard, Ada Language Reference Manual (ARM), ANSI/MIL-STD-1815A (22 January 1983), supersedes MIL-STD-1815 (10 December 1980).
Official description of the Ada language.

[WICH82] Wichmann, B.A.; J.C.D. Nissen; et al., Ada-Europe Guidelines for Ada Compiler Specification and Selection, Ada Letters, III-1 (October 1982) pp. 37-50.
Provides a list of questions to be used in selection of Ada compilers, used as primary basis for this paper.

ACRONYMS

ACEC	Ada Compiler Evaluation Capability
ACVC	Ada Compiler Validation Capability
AJPO	Ada Joint Program Office
ARM	Ada Language Reference Manual, ANSI/MIL-STD-1815A
DOD	Department of Defense
ECSPO	Embedded Computer Standardization Program Office
HOL	High Order Language
VDD	Version Description Document

END

8-87

DTIC